Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Organización de Lenguajes y Compiladores 1

Sección B

Fecha: 29/04/2022

MANUAL TÉCNICO

Nombre:	Carné:
Gerson Rubén Quiroa del Cid	2020 00166

Índice

Requisitos del sistema

Para la instalación de la aplicación, su computadora y/o laptop debe cumplir como mínimo los siguientes requerimientos:

Para Windows

- Windows Vista SP2 (8u51 y superiores)
- Windows Server 2008 R2 SP1 (64 bits)
- Windows Server 2012 y 2012 R2 (64 bits)
- RAM: 128 MB
- Procesador: Mínimo Pentium 2 a 266 MHz
- Exploradores: Internet Explorer 9 y superior, Firefox

Mac OS

- Mac con Intel que ejecuta Mac OS X 10.8.3+, 10.9+
- Privilegios de administrador para la instalación
- Explorador de 64 bits
- Se requiere un explorador

Linux

- Oracle Linux 7.x (64 bits)2(8u20 y superiores)
- Red Hat Enterprise Linux 7.x (64 bits)2(8u20 y superiores)
- Suse Linux Enterprise Server 12.x (64 bits)2(8u31 y superiores)
- Ubuntu Linux 12.04 LTS, 13.x
- Ubuntu Linux 15.10 (8u65 y superiores)

IMPORTANTE: Independientemente del sistema operativo del usuario, es necesario que previamente instale Python en su computadora.

Editor utilizado: Visual Studio Code IDE 8.2

Versión de JavaScript utilizado: 3.9.4

Introducción

El manual técnico tiene como finalidad de explicar el funcionamiento de cada función utilizado en el presente programa, con el objetivo de que no haya confusión al leer el código del programa y cualquier persona con conocimiento básico de programación en JavaScript pueda comprender lo que se hizo en cada línea de código. Por lo cual se recomienda que toda persona que quiera leer o modificar el código vea este manual para ahorraste tiempo y así sea más fácil su comprensión.

Backend

Esta parte de nuestro proyecto es donde está toda la lógica del programa, donde se reciben las solicitudes de frontend, se procesan y se envía una respuesta. En este caso el principal servicio es traducir código establecido.

Index.js (Main)

Este módulo es nuestro servicio api, donde se reciben todas las solicitudes del frontend. Se utilizó la librería de express para crear dicho servicio. Este módulo cuenta con las siguientes peticiones: get (“/”, “/errores”, “/ast”, “/símbolo”) y post (“/grammar”).

```
16  const app = express();
17  app.use(express.json());
18  app.use(express.urlencoded({ extended: true }));
19  app.use(cors());
20
21  app.get("/", (req, res) => {
22    res.send("<h1>Hello World!</h1>");
23  });
24
25  app.post("/grammar", (req, res) => {
26    // limpiando la lista de errores
27    lista_errores = [];
28    const data = req.body;
29    console.log("soy backend");
30    let salida = "";
31    let sinErrores = true;
32    const ast = parser.parse(data.data);
33    console.log(ast);
34    const scope = new Scope(null);
35    // primera pasada: guardando todas las funciones declaradas
36    for (const instr of ast) {
37      try {
38        if (instr instanceof Funcion) {
39          instr.ejecutar(scope);
40        }
41      } catch (error) {
42        sinErrores = false;
```

Interprete

En esta carpeta está toda la lógica del intérprete del programa. Tanto la gramática generada por jison así como toda la lógica semántica del programa.

grammar.jison

Gramática que genera el intérprete en lenguaje Javascript.

```
ini
    : instrucciones EOF {
        return $1;
    }
    | EOF
;

instrucciones
    : instrucciones instruccion { $1.push($2); $$ = $1; }
    | instruccion               { $$ = [$1]; }
;

instruccion
    : declaracion PUNTO_COMA
    | asignacion_simple PUNTO_COMA
    | imprimir PUNTO_COMA
    | incremento_decremento PUNTO_COMA
    | declaracion_vector PUNTO_COMA
    | asignacion_simple_vector PUNTO_COMA
    | if
    | while
    | switch
    | for
    | do-while
    | funcion
    | metodo
```

ast.jison

Esta gramática se utilizó para poder generar el ast y poder así graficarlo.

```
ini
    : instrucciones EOF {   $$ = new Nodo('inicio');
                            $$.agregarHijo($1);
                            return $$;
                        }
    | EOF
;

instrucciones
    : instrucciones instruccion {   $$ = new Nodo('instrucciones');
                                    $$.hijos = [...$1.hijos, ...$2.hijos];
                                }
    | instruccion               {   $$ = new Nodo('instrucciones');
                                    $$.hijos = [...$1.hijos];
                                }
    | error                     { throw new _Error(@1.first_line, @1.first_column, "Sintáctico", "No se esperaba el siguiente token: " + yytext
;

instruccion
    : declaracion PUNTO_COMA                { $$ = new Nodo('instruccion'); $$.agregarHijo($1); $$.agregarHijo(new Nodo(';')); }
    | asignacion_simple PUNTO_COMA          { $$ = new Nodo('instruccion'); $$.agregarHijo($1); $$.agregarHijo(new Nodo(';')); }
    | imprimir PUNTO_COMA                   { $$ = new Nodo('instruccion'); $$.agregarHijo($1); $$.agregarHijo(new Nodo(';')); }
    | incremento_decremento PUNTO_COMA      { $$ = new Nodo('instruccion'); $$.agregarHijo($1); $$.agregarHijo(new Nodo(';')); }
    | declaracion_vector PUNTO_COMA         { $$ = new Nodo('instruccion'); $$.agregarHijo($1); $$.agregarHijo(new Nodo(';')); }
    | asignacion_simple_vector PUNTO_COMA   { $$ = new Nodo('instruccion'); $$.agregarHijo($1); $$.agregarHijo(new Nodo(';')); }
    | if                                    { $$ = new Nodo('instruccion'); $$.agregarHijo($1); }
    | while                                 { $$ = new Nodo('instruccion'); $$.agregarHijo($1); }
    | switch                                { $$ = new Nodo('instruccion'); $$.agregarHijo($1); }
```

Acceso.ts

El acceso nos permite acceder a las variables y obtener tanto su valor como su tipo.

```ts
import { _Error } from "../Error/_Error";
import { Scope } from "../Extra/Scope";
import { Expresion } from "./Expresion";
import { Retorno } from "./Retorno";

export class Acceso extends Expresion {
    constructor(public id: string, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        const value = scope.getValor(this.id, this.linea, this.columna);
        if(value != null) {
            return {value: value.valor, type: value.type, output: null}
        }
        throw new _Error(this.linea, this.columna, 'Semántico', 'No se ha declarado la variable ' + value.id);
    }
}
```

AccesoVector.ts

Al igual que el acceso, el acceso vector puede acceder a los vectores declarados y poder así obtener el valor y el tipo de una posición.

```
export class AccesoVector extends Expresion {
    constructor(public id: string, private dimensiones: Expresion[], linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // devuelve la lista entera
        const lista = scope.getValor(this.id, this.linea, this.columna);
        // obtenemos las dimensionales de la lista y la posicion que queremos
        let dimension = [];
        for (const d of this.dimensiones) {
            let value = d.ejecutar(scope);
            if (value.type != Tipo.ENTERO) {
                throw new _Error(this.linea, this.columna, "Semántico", "No se puede acceder a la posicion " + value.value);
            }
            dimension.push(value.value);
        }
        let valor = this.devolverValor(lista, dimension);
        return {value: valor, type: lista.type, output: null}
    }

    public devolverValor(lista: Simbolo, dimension: number[]): any {
        if (lista != null) {
            if (dimension.length == 1) {
                if (lista.valor.length <= dimension[0]) {
                    throw new _Error(this.linea, this.columna, 'Semántico', 'No existe la posición [' + dimension[0] + '].');
```

Aritmetica.ts

Esta clase es la encargada de hacer las operaciones aritméticas con las que cuenta nuestro lenguaje de programación.

```
export class Aritmetica extends Expresion {
    constructor(private izquierda: Expresion | LlamadaFuncion, priva
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // valores de las expresiones
        let valorIzquierda: Retorno;
        let valorDerecha: Retorno;
        let flagIzq: string = null, flagDer: string = null;
        if (this.izquierda instanceof LlamadaFuncion) {
            const funIzquierda = this.izquierda.ejecutar(scope);
            console.log("output:: " + funIzquierda.retorno.output)
            if (funIzquierda.retorno != null) {
                valorIzquierda = funIzquierda.retorno;
                if (funIzquierda.output != null) {
                    flagIzq = funIzquierda.output;
                }
            }
        } else {
            valorIzquierda = this.izquierda.ejecutar(scope);
        }
        if (this.derecha instanceof LlamadaFuncion) {
            const funDerecha = this.derecha.ejecutar(scope);
            if (funDerecha.retorno != null) {
                valorDerecha = funDerecha.retorno;
```

Casteos.ts

Esta clase es la encargada de castear algunos valores de un tipo de datos a otros. Dependiendo si es permitido hacer dicho casteo.

```
export class Casteo extends Expresion {
    constructor(private tipo: string, private expresion: Expresion, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // valor de la expresion
        const valorExpresion = this.expresion.ejecutar(scope);

        if (valorExpresion.type == Tipo.ENTERO && this.tipo == 'double') {
            return { value: valorExpresion.value.toFixed(1), type: Tipo.DECIMAL, output: valorExpresion.output };
        } else if (valorExpresion.type == Tipo.DECIMAL && this.tipo == 'int') {
            return { value: parseInt(valorExpresion.value, 10), type: Tipo.ENTERO, output: valorExpresion.output };
        } else if (valorExpresion.type == Tipo.ENTERO && this.tipo == 'char') {
            return { value: String.fromCharCode(valorExpresion.value), type: Tipo.CARACTER, output: valorExpresion.output };
        } else if (valorExpresion.type == Tipo.CARACTER && this.tipo == 'int') {
            return { value: valorExpresion.value.charCodeAt(0), type: Tipo.ENTERO, output: valorExpresion.output };
        } else if (valorExpresion.type == Tipo.CARACTER && this.tipo == 'double') {
            return { value: valorExpresion.value.charCodeAt(0).toFixed(1), type: Tipo.DECIMAL, output: valorExpresion.output };
        }
        throw new _Error(this.linea, this.columna, "Semántico", "Error de casteo. No es posible pasar de " + Tipo[valorExpresion.type] + " a " +
    }
}
```

Expresion.ts

Esta clase es una plantilla que utiliza todas las clases que extienden de ella. Por lo que esta clase la creamos abstracta.

```
import { Scope } from "../Extra/Scope";
import * as retorno from "./Retorno";

export abstract class Expresion {
    public linea: number;
    public columna: number;

    constructor(linea: number, columna: number) {
        this.linea = linea;
        this.columna = columna;
    }

    public abstract ejecutar(scope: Scope): retorno.Retorno;

    public tipoDominanteSuma(tipo1: retorno.Tipo, tipo2: retorno.Tipo): retorno.Tipo {
        return retorno.dominanteSuma[tipo1][tipo2];
    }
    public tipoDominanteResta(tipo1: retorno.Tipo, tipo2: retorno.Tipo): retorno.Tipo {
        return retorno.dominanteResta[tipo1][tipo2];
    }
    public tipoDominanteMultiplicacion(tipo1: retorno.Tipo, tipo2: retorno.Tipo): retorno.Tipo {
        return retorno.dominanteMultiplicacion[tipo1][tipo2];
```

IncrementoDeclemento.ts

Esta clase es la encargada de incrementar o decrementar una variable en uno.

```
export class IncrementoDecremento extends Expresion {
    constructor(private expresion: Acceso, private tipo: TipoAritmetica, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // valor de la expresion
        const valorExpresion = this.expresion.ejecutar(scope);
        let valueDespues;
        if (this.tipo == TipoAritmetica.SUMA) {
            if (valorExpresion.type == Tipo.ENTERO || valorExpresion.type == Tipo.DECIMAL) {
                scope.setValor(this.expresion.id, valorExpresion.value + 1, valorExpresion.type, this.linea, this.columna);
                return { value: valorExpresion.value, type: valorExpresion.type, output: valorExpresion.output };
            }
            throw new _Error(this.linea, this.columna, "Semántico", "No se puede incrementar el siguiente valor: " + valorExpresion.value);
        } else {
            if (valorExpresion.type == Tipo.ENTERO || valorExpresion.type == Tipo.DECIMAL) {
                scope.setValor(this.expresion.id, valorExpresion.value - 1, valorExpresion.type, this.linea, this.columna);
                return { value: valorExpresion.value, type: valorExpresion.type, output: valorExpresion.output };
            }
            throw new _Error(this.linea, this.columna, "Semántico", "No se puede decrementar el siguiente valor: " + valorExpresion.value);
        }
    }
}
```

Length.ts

Esta clase es la encargada de devolver el tamaño de un vector o una cadena.

```
export class Length extends Expresion {
    constructor(private expresion: Expresion, linea:number, columna:number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutando la expresion
        const val = this.expresion.ejecutar(scope);
        if (!(val.value instanceof Array) && val.type != Tipo.CADENA) {
            throw new _Error(this.linea, this.columna, "Semántico", "Se esperaba un VECTOR, LISTA o CADENA y se obtuvo un " + Tipo[val.type]);
        }
        return { value: val.value.length, type: Tipo.ENTERO, output: val.output };
    }
}
```

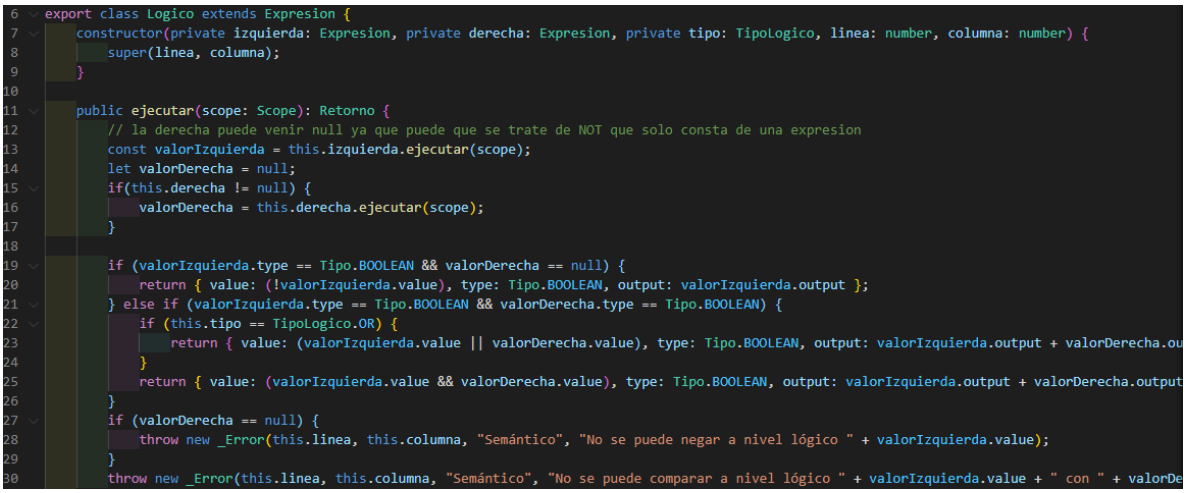
Literal.ts

Esta clase es la encargada de convertir los enteros, cadenas, decimales, booleanos, etc. en su respectivo tipo de dato.

```
export class Literal extends Expresion {
    constructor(private value: any, private tipo: TipoLiteral, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        if (this.tipo == TipoLiteral.ENTERO) {
            return { value: Number(this.value), type: Tipo.ENTERO, output: null };
        } else if (this.tipo == TipoLiteral.DECIMAL) {
            return { value: Number(this.value), type: Tipo.DECIMAL, output: null };
        } else if (this.tipo == TipoLiteral.BOOLEAN) {
            if (this.value.toString().toLowerCase() == "true") {
                return { value: true, type: Tipo.BOOLEAN, output: null };
            }
            return { value: false, type: Tipo.BOOLEAN, output: null };
        } else if (this.tipo == TipoLiteral.CARACTER) {
            return { value: this.value.toString(), type: Tipo.CARACTER, output: null };
        } else if (this.tipo == TipoLiteral.CADENA) {
            return { value: this.value.toString(), type: Tipo.CADENA, output: null };
        }
        throw new _Error(this.linea, this.columna, "Semántico", "Error");
    }
}
```

Logico.ts

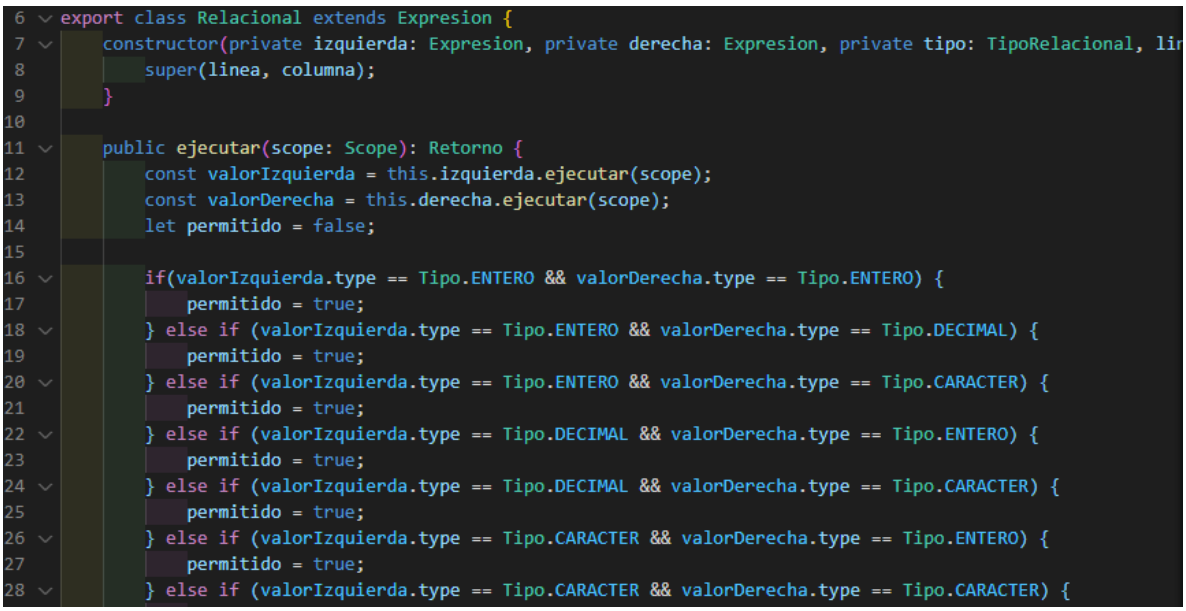
Esta clase es la encargada de concatenar a nivel lógico AND, OR o NOT.



```
export class Logico extends Expresion {
    constructor(private izquierda: Expresion, private derecha: Expresion, private tipo: TipoLogico, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // la derecha puede venir null ya que puede que se trate de NOT que solo consta de una expresion
        const valorIzquierda = this.izquierda.ejecutar(scope);
        let valorDerecha = null;
        if(this.derecha != null) {
            valorDerecha = this.derecha.ejecutar(scope);
        }

        if (valorIzquierda.type == Tipo.BOOLEAN && valorDerecha == null) {
            return { value: (!valorIzquierda.value), type: Tipo.BOOLEAN, output: valorIzquierda.output };
        } else if (valorIzquierda.type == Tipo.BOOLEAN && valorDerecha.type == Tipo.BOOLEAN) {
            if (this.tipo == TipoLogico.OR) {
                return { value: (valorIzquierda.value || valorDerecha.value), type: Tipo.BOOLEAN, output: valorIzquierda.output + valorDerecha.ou
            }
            return { value: (valorIzquierda.value && valorDerecha.value), type: Tipo.BOOLEAN, output: valorIzquierda.output + valorDerecha.output
        }
        if (valorDerecha == null) {
            throw new _Error(this.linea, this.columna, "Semántico", "No se puede negar a nivel lógico " + valorIzquierda.value);
        }
        throw new _Error(this.linea, this.columna, "Semántico", "No se puede comparar a nivel lógico " + valorIzquierda.value + " con " + valorDe
```

Relacional.ts

Esta clase es la encargada de comparar 2 expresiones y retornar un booleano dependiendo si se cumple la condición o no.



```
export class Relacional extends Expresion {
    constructor(private izquierda: Expresion, private derecha: Expresion, private tipo: TipoRelacional, lir
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        const valorIzquierda = this.izquierda.ejecutar(scope);
        const valorDerecha = this.derecha.ejecutar(scope);
        let permitido = false;

        if(valorIzquierda.type == Tipo.ENTERO && valorDerecha.type == Tipo.ENTERO) {
            permitido = true;
        } else if (valorIzquierda.type == Tipo.ENTERO && valorDerecha.type == Tipo.DECIMAL) {
            permitido = true;
        } else if (valorIzquierda.type == Tipo.ENTERO && valorDerecha.type == Tipo.CARACTER) {
            permitido = true;
        } else if (valorIzquierda.type == Tipo.DECIMAL && valorDerecha.type == Tipo.ENTERO) {
            permitido = true;
        } else if (valorIzquierda.type == Tipo.DECIMAL && valorDerecha.type == Tipo.CARACTER) {
            permitido = true;
        } else if (valorIzquierda.type == Tipo.CARACTER && valorDerecha.type == Tipo.ENTERO) {
            permitido = true;
        } else if (valorIzquierda.type == Tipo.CARACTER && valorDerecha.type == Tipo.CARACTER) {
```

Relacional.ts

Esta clase es la encargada de retornar la salida, el valor y el tipo cuando es necesario.

```typescript
export enum Tipo {
    ENTERO,
    DECIMAL,
    BOOLEAN,
    CARACTER,
    CADENA,
    ERROR
}

export type Retorno = {
    value: any,
    type: Tipo
    output: string;
}

export const dominanteSuma = [
    [Tipo.ENTERO, Tipo.DECIMAL, Tipo.ENTERO, Tipo.ENTERO, Tipo.CADENA],
    [Tipo.DECIMAL, Tipo.DECIMAL, Tipo.DECIMAL, Tipo.DECIMAL, Tipo.CADENA],
    [Tipo.ENTERO, Tipo.DECIMAL, Tipo.ERROR, Tipo.ERROR, Tipo.CADENA],
    [Tipo.ENTERO, Tipo.DECIMAL, Tipo.ERROR, Tipo.CADENA, Tipo.CADENA],
    [Tipo.CADENA, Tipo.CADENA, Tipo.CADENA, Tipo.CADENA, Tipo.CADENA]
];

export const dominanteResta = [
    [Tipo.ENTERO, Tipo.DECIMAL, Tipo.ENTERO, Tipo.ENTERO, Tipo.ERROR],
    [Tipo.DECIMAL, Tipo.DECIMAL, Tipo.DECIMAL, Tipo.DECIMAL, Tipo.ERROR],
    [Tipo.ENTERO, Tipo.DECIMAL, Tipo.ERROR, Tipo.ERROR, Tipo.ERROR],
    [Tipo.ENTERO, Tipo.DECIMAL, Tipo.ERROR, Tipo.ERROR, Tipo.ERROR],
    [Tipo.ERROR, Tipo.ERROR, Tipo.ERROR, Tipo.ERROR, Tipo.ERROR]
];
```

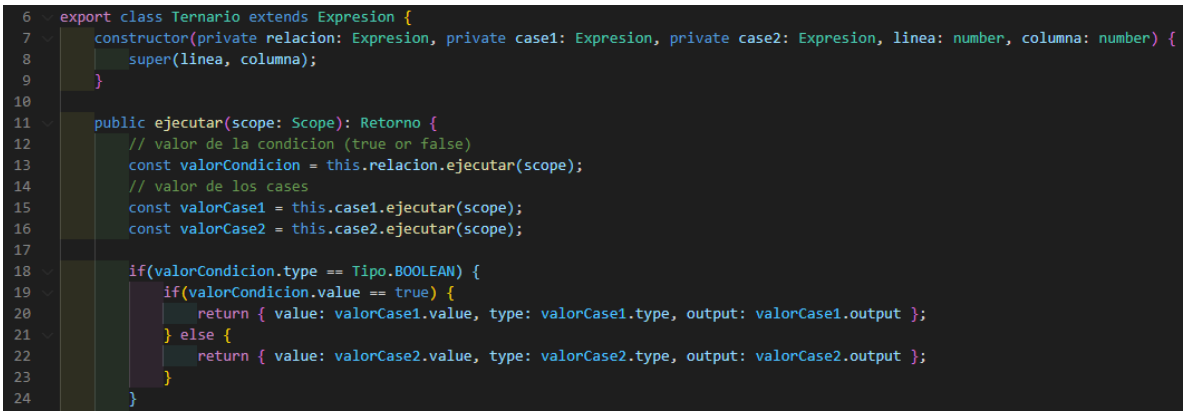
Round.ts

Esta clase es la encargada de redondear valores decimales.

```typescript
export class Round extends Expresion {
    constructor(private expresion: Expresion, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutamos la expresion
        const val = this.expresion.ejecutar(scope);
        if(val.type != Tipo.DECIMAL) {
            throw new _Error(this.linea, this.columna, "Semántico", "Se esperaba un DECIMAL y se obtuvo un " + Tipo[val.type]);
        }
        return { value: Math.round(val.value), type: Tipo.ENTERO, output: val.output };
    }
}
```

Ternario.ts

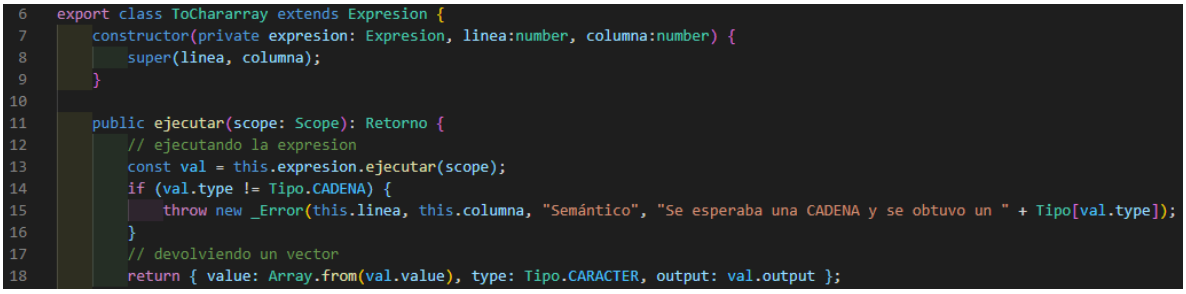
Esta clase es la encargada de hacer un if resumido. Si se cumple la condición devuelve el primer valor si no, el segundo.



```
export class Ternario extends Expresion {
    constructor(private relacion: Expresion, private case1: Expresion, private case2: Expresion, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // valor de la condicion (true or false)
        const valorCondicion = this.relacion.ejecutar(scope);
        // valor de los cases
        const valorCase1 = this.case1.ejecutar(scope);
        const valorCase2 = this.case2.ejecutar(scope);

        if(valorCondicion.type == Tipo.BOOLEAN) {
            if(valorCondicion.value == true) {
                return { value: valorCase1.value, type: valorCase1.type, output: valorCase1.output };
            } else {
                return { value: valorCase2.value, type: valorCase2.type, output: valorCase2.output };
            }
        }
```

ToChararray.ts

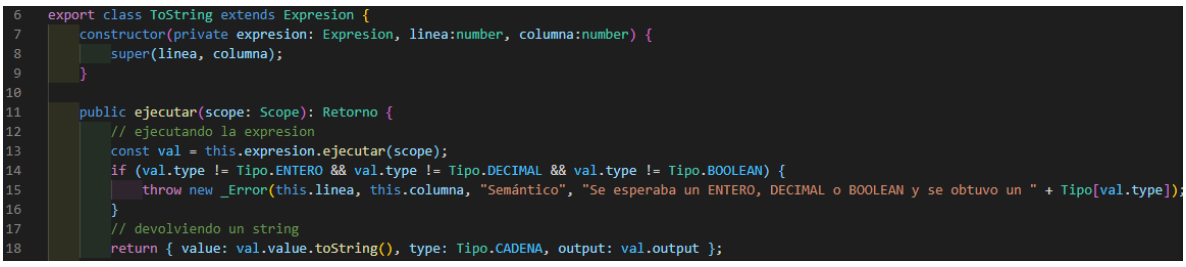
Esta clase es la encargada de convertir una cadena en una lista de caracteres con dicha cadena.



```
export class ToChararray extends Expresion {
    constructor(private expresion: Expresion, linea:number, columna:number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutando la expresion
        const val = this.expresion.ejecutar(scope);
        if (val.type != Tipo.CADENA) {
            throw new _Error(this.linea, this.columna, "Semántico", "Se esperaba una CADENA y se obtuvo un " + Tipo[val.type]);
        }
        // devolviendo un vector
        return { value: Array.from(val.value), type: Tipo.CARACTER, output: val.output };
```

ToString.ts

Esta clase es la encargada de convertir un tipo de dato a una cadena.



```
export class ToString extends Expresion {
    constructor(private expresion: Expresion, linea:number, columna:number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutando la expresion
        const val = this.expresion.ejecutar(scope);
        if (val.type != Tipo.ENTERO && val.type != Tipo.DECIMAL && val.type != Tipo.BOOLEAN) {
            throw new _Error(this.linea, this.columna, "Semántico", "Se esperaba un ENTERO, DECIMAL o BOOLEAN y se obtuvo un " + Tipo[val.type]);
        }
        // devolviendo un string
        return { value: val.value.toString(), type: Tipo.CADENA, output: val.output };
```

ToUpLowCase.ts

Esta clase es la encargada de convertir todo en mayúsculas o minúsculas de una cadena dependiendo sea el caso.

```typescript
// estas clases estarán en las expresiones ya que devuelve una expresion
export class ToUpperCase extends Expresion {
    constructor(private cadena: Expresion, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutamos la expresion
        const value = this.cadena.ejecutar(scope);
        if (value.type != Tipo.CADENA) {
            throw new _Error(this.linea, this.columna, "Semántico", "Se esperaba una CADENA y se obtuvo un " + Tipo[value.type]);
        }
        return { value: value.value.toUpperCase(), type: Tipo.CADENA, output: value.output };
    }
}

export class ToLowerCase extends Expresion {
    constructor(private cadena: Expresion, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutamos la expresion
        const value = this.cadena.ejecutar(scope);
        if (value.type != Tipo.CADENA) {
            throw new _Error(this.linea, this.columna, "Semántico", "Se esperaba una CADENA y se obtuvo un " + Tipo[value.type]);
```

TypeOf.ts

Esta clase es la encargada de devolver el tipo de dato de una expresión como cadena.

```typescript
export class TypeOf extends Expresion {
    constructor(private expresion: Expresion, linea:number, columna:number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutando la expresion
        const val = this.expresion.ejecutar(scope);
        if (val.value instanceof Array) {
            return { value: "VECTOR", type: Tipo.CADENA, output: val.output };
        }
        // devolviendo el tipo de dato
        return { value: Tipo[val.type].toString(), type: Tipo.CADENA, output: val.output };
    }
}
```

ArbolAST.ts

Esta clase es la encargada de recorrer y generar el dot para la gráfica del ast.

```
export class ArbolAST {
    constructor() {
        id = 0;
    }

    public recorrerArbol(nodo: Nodo): string {
        let salida: string;
        if (nodo != null) {
            if (nodo.id == 0) {
                nodo.id = id;
                id++;
            }
            salida = nodo.id + '[label="' + nodo.valor + '" shape="circle"];';
            nodo.hijos.forEach(element => {
                salida += nodo.id + '->' + id + ';';
                salida += this.recorrerArbol(element);
            });
            return salida;
        }
        return "";
```

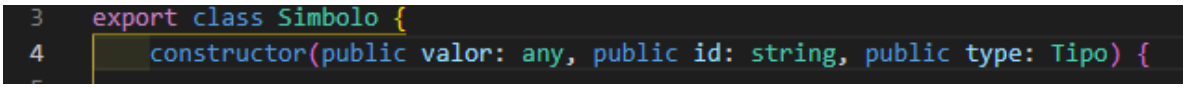
Scope.ts

Esta clase es la encargada de guardar las variables del entorno actual, así como las funciones. También devuelven los valores cuando sea necesario.

```
export class Scope {
    public variables: Map<string, Simbolo>;
    public funciones: Map<string, Funcion>;
    public simbolos = [];

    constructor(public padre: Scope | null) {
        this.variables = new Map();
        this.funciones = new Map();
    }

    public crearVar(id: string, value: any, type: Tipo, linea: number, columna: number) {
        let scope: Scope | null = this;

        while(scope != null) {
            if(scope.variables.has(id)) {
                throw new _Error(linea, columna, "Semántico", "La variable " + id + " ya ha sido declarada.
            }
            scope = scope.padre;
        }
        if(value == null) {
            if(type == Tipo.ENTERO) {
                this.variables.set(id, new Simbolo(0, id, type));
            } else if(type == Tipo.DECIMAL) {
                this.variables.set(id, new Simbolo((0).toFixed(1), id, type));
            } else if(type == Tipo.CADENA) {
                this.variables.set(id, new Simbolo("", id, type));
            } else if(type == Tipo.CARACTER) {
                this.variables.set(id, new Simbolo('\u0000', id, type));
            } else if(type == Tipo.BOOLEAN) {
                this.variables.set(id, new Simbolo(true, id, type));
            }
```

Simbolo.ts

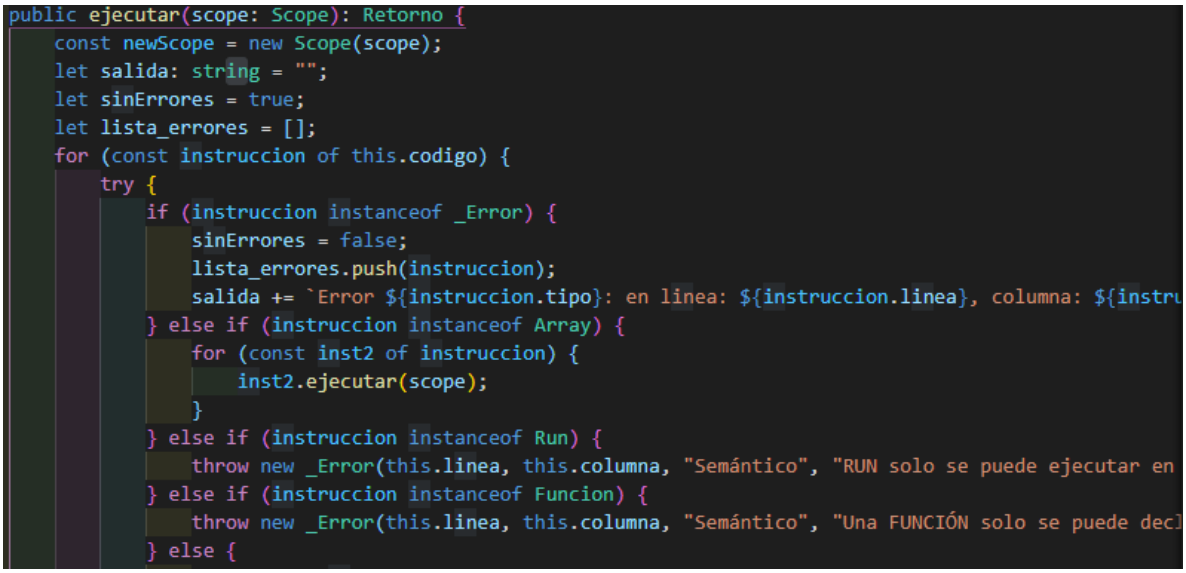
Esta clase es la encargada de crear los símbolos del programa.



```
3   export class Simbolo {
4       constructor(public valor: any, public id: string, public type: Tipo) {
```

Bloque.ts

Esta clase es la encargada de ejecutar un bloque de código encerrada por {}.



```
public ejecutar(scope: Scope): Retorno {
    const newScope = new Scope(scope);
    let salida: string = "";
    let sinErrores = true;
    let lista_errores = [];
    for (const instruccion of this.codigo) {
        try {
            if (instruccion instanceof _Error) {
                sinErrores = false;
                lista_errores.push(instruccion);
                salida += `Error ${instruccion.tipo}: en linea: ${instruccion.linea}, columna: ${instru
            } else if (instruccion instanceof Array) {
                for (const inst2 of instruccion) {
                    inst2.ejecutar(scope);
                }
            } else if (instruccion instanceof Run) {
                throw new _Error(this.linea, this.columna, "Semántico", "RUN solo se puede ejecutar en
            } else if (instruccion instanceof Funcion) {
                throw new _Error(this.linea, this.columna, "Semántico", "Una FUNCIÓN solo se puede dec
            } else {
```

Declaracion.ts

Esta clase es la encargada de la declaración de variables.

```
export class Declaracion extends Instruccion {
    constructor(public type: string, public ids: string[], private value: Expresion | LlamadaFuncion, linea
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        console.log("tipo: " + this.type)
        let tipo;
        if (this.type == "int") {
            tipo = Tipo.ENTERO;
        } else if (this.type == "double") {
            tipo = Tipo.DECIMAL;
        } else if (this.type == "boolean") {
            tipo = Tipo.BOOLEAN;
        } else if (this.type == "char") {
            tipo = Tipo.CARACTER;
        } else if (this.type == "string") {
            tipo = Tipo.CADENA;
        }

        for (const id of this.ids) {
            if (this.value != null) {
                if (this.value instanceof Expresion) {
                    // si es una expresion
                    const val = this.value.ejecutar(scope);
                    if (val.value instanceof Array) {
                        throw new _Error(this.linea, this.columna, "Semántico", "Tipos incompatibles. No s
```

DeclaracionVector.ts

Esta clase es la encargada de declarar vectores de 1 y 2 dimensiones.

```
export class DeclaracionVector1 extends Instruccion {
    constructor(private type1: string, private id: string, private type2: string, private dimensiones: Expr
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        let dimension = [];
        for (const d of this.dimensiones) {
            let value = d.ejecutar(scope);
            if (value.type != Tipo.ENTERO) {
                throw new _Error(this.linea, this.columna, "Semántico", "No se puede crear una dimensión cc
            }
            dimension.push(value);
        }
        // obteniendo la lista creada (uno o dos dimensiones)
        let lista = this.crearLista(dimension);
        let tipo = this.tipo();
        scope.crearVar(this.id, lista, tipo, this.linea, this.columna);
        console.log("value: ")
```

DoWhile.ts

Esta clase es la encargada de recrear la función do-while.

```
export class DoWhile extends Instruccion {
    constructor(private bloque: Instruccion, private condicion: Expresion, linea: number, columna: number)
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // esta vez con let, ya que la ejecutaremos varias veces hasta que deje de cumplirse la condicion
        let value;
        let salida: string = "";
        // ejecutando el bloque por lo menos una vez
        do {
            const retorno = this.bloque.ejecutar(scope);
            // verificamos si hay sentencias de transferencias
            if (retorno != null && retorno != undefined) {
                if (retorno.transferencia != null) {
                    if (retorno.transferencia.type == TipoTransferencia.BREAK) {
                        if (retorno.output != null) {
                            salida += retorno.output;
```

For.ts

Esta clase es la encargada de recrear la función for.

```
export class For extends Instruccion {
    constructor(private id: Instruccion, private condicion: Expresion, private actualizacion: Instruccion
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // un nuevo scope
        const newScope = new Scope(scope);
        // ejecutamos la declaracion o asignacion
        this.id.ejecutar(newScope);
        // ejecutamos la condicion
        let valCondicion = this.condicion.ejecutar(newScope);
        // comprobamos si nos devuelve un boolean
        if (valCondicion.type != Tipo.BOOLEAN) {
            throw new _Error(this.linea, this.columna, "Semántico", "La condición a evaluar tiene que retor
        }
```

Función.ts

Esta clase es la encargada de crear y guardar las funciones y métodos.

```
export class Funcion extends Instruccion {
    constructor(private id: string, public parametros: Declaracion[], private tipoRetorno: string | null, p
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        let tipo = 'Método';
        let tipoDato = 'VOID';
        if (this.tipoRetorno != null) {
            tipo = 'Función';
            tipoDato = this.tipoRetorno.toUpperCase();
        }
        scope.guardarFuncion(this.id, this, this.linea, this.columna, tipo, tipoDato);
        return null;
    }
```

IfElse.ts

Esta clase es la encargada de recrear la instrucción if-else.

```
export class IfElse extends Instruccion {
    constructor (private condicion: Expresion, private bloque: Instruccion, private bloqueElse: Instruccior
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutamos y obtenemos el valor de la condición
        const valueCondicion = this.condicion.ejecutar(scope);

        // verificamos que se trate de un boolean, sino lanzamos un error
        if (valueCondicion.type != Tipo.BOOLEAN) {
            throw new _Error(this.linea, this.columna, "Semántico", "La condición a evaluar tiene que retor
        }
```

Imprimir.ts

Esta clase es la encargada de recrear la instrucción print y prinln.

```
export class Println extends Instruccion {
    constructor(private value: Expresion, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        const val = this.value.ejecutar(scope);
        // aki hay que poner el print en la pagina web, mientras solo le dearemos un console.log
        console.log(val.value);
        let value = val.value.toString();
        const salto = '\\n', barraInv = '\\\\', comillaD = '\\\"', tab = '\\t', comillaS = "\\\'";
        if (value.includes(salto)) {
            value = value.replace(salto, '\n');
```

Instrucción.ts

Esta clase es la plantilla para todas las clases que extienden de esta clase. Esta clase es de tipo abstracta.

```
4  export abstract class Instruccion {
5      public linea: number;
6      public columna: number;
7
8      constructor(linea: number, columna: number) {
9          this.linea = linea;
10         this.columna = columna;
11     }
12
13     public abstract ejecutar(scope: Scope): Retorno;
14 }
```

LlamadaFuncion.ts

Esta clase es la encargada de recrear la instrucción de una llamada a una función.

```
export class LlamadaFuncion extends Instruccion {
    constructor(private id: string, private argumentos: Expresion[] | null, linea:number, columna:number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // obtenemos la función (si se encuentra)
        const funcion = scope.getFuncion(this.id, this.linea, this.columna);
        if (this.argumentos.length != funcion.parametros.length) {
            // si los parámetros no coinciden con la cantidad de argumentos es un error semántico
            throw new _Error(this.linea, this.columna, "Semántico", "La cantidad de argumentos es errónea."
        }
        // comprobamos si los argumentos que nos mandan son del tipo de los parametros
        funcion.tipoArgumentos(this.argumentos, scope)
        // obteniendo el scope global y creando el nuevo scope para la funcion
        const newScope = new Scope(scope.getGlobal());
        for (let i = 0; i < this.argumentos.length; i++) {
            // argumentos de la llamada a la función
            const valueArgs = this.argumentos[i].ejecutar(scope);
            newScope.crearVar(funcion.parametros[i].ids[0], valueArgs.value, valueArgs.type, this.linea, th
        }
```

Retorno.ts

Esta clase es la encargada de retornar expresiones cuando se trate de returns y las salidas de los prints.

```
4  export type Retorno = {
5      output: string,
6      transferencia: transferencia
7      retorno: Retorno2
8  }
```

Run.ts

Esta clase es la encargada de iniciar la lógica del programa. Parecido al método main.

```
export class Run extends Instruccion {
    constructor (private funcion: Instruccion, linea: number, columna: number) {
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        return this.funcion.ejecutar(scope);
    }
```

Switch.ts

Esta clase es la encargada de recrear la instrucción switch.

```
export class Switch extends Instruccion {
    constructor(private condicion: Expresion, private case_list: Case[], private _default: Default, linea:
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        let salida: string = "";
        let sinBreak = false, usarDefault = true;
        if (this.case_list != null) {
            // cases del switch
            for (const _case of this.case_list) {
                // verificando si se cumple la igualdad del switch con el case y si son compatibles
                const comparacion = new Relacional(this.condicion, _case.comparar(), TipoRelacional.IGUAL,
                if (comparacion.value || sinBreak) {
                    usarDefault = false;
                    // ejecutando las instrucciones del case
                    const resultado = _case.ejecutar(scope);

                    if (resultado != null && resultado != undefined) {
                        if (resultado.transferencia != null) {
                            if (resultado.transferencia.type == TipoTransferencia.BREAK) {
                                if (resultado.output != null) {
                                    salida += resultado.output;
                                }
                                // como encontro el break, se saldrá del switch y no ejecutará el default
                                sinBreak = false;
```

Transferencias.ts

Esta clase es la encargada de recrear las instrucciones de break, continue y return.

```
export class Break extends Instruccion {
    constructor (linea: number, columna: number) {
        super(linea, columna);
    }
```

```
export class Continue extends Instruccion {
    constructor (linea: number, columna: number) {
        super(linea, columna);
    }
```

```
export class Return extends Instruccion {
    constructor(private expresion: Expresion | LlamadaFuncion, linea:number, columna:number) {
        super(linea, columna);
    }
```

While.ts

Esta clase es la encargada de recrear las instrucciones while.

```
export class While extends Instruccion {
    constructor(private condicion: Expresion, private bloque: Instruccion, linea: number, columna: number)
        super(linea, columna);
    }

    public ejecutar(scope: Scope): Retorno {
        // ejecutamos la condición
        // esta vez con let, ya que la ejecutaremos varias veces hasta que deje de cumplirse la condicion
        let value = this.condicion.ejecutar(scope);
        // comprobamos si nos devuelve un boolean
        if (value.type != Tipo.BOOLEAN) {
            throw new _Error(this.linea, this.columna, "Semántico", "La condición a evaluar tiene que retor
        }
        let salida: string = "";
        // mientras value.value == true
        while (value.value) {
```

Frontend (webapp)

En esta carpeta se encuentra el frontend de nuestra aplicación web. Donde se tienen varios componentes para poder navegar entre la aplicació

Función analizar:

Esta función es la que contiene el autómata que reconoce cada carácter que haya dentro del archivo de entrada, si no encuentra el carácter, lo mandará a una lista de errores.

```jsx
import "./App.css";
import {
  BrowserRouter as Router,
  Routes,
  Route
} from "react-router-dom";
import Inicio from "./components/Inicio";
import ErrorPage from "./components/ErrorPage";
import ErroresReport from "./components/ErroresReport";
import ASTReport from "./components/ASTReport";
import SimbolosReport from "./components/SimbolosReport";

function App() {
  return (
    <Router>
      <Routes>
        <Route path='/' element={<Inicio />} />
        <Route path='/errores' element={<ErroresReport />} />
        <Route path='/AST' element={<ASTReport />} />
        <Route path='/simbolos' element={<SimbolosReport />} />
        <Route path='*' element={<ErrorPage />} />
      </Routes>
    </Router>

  );
}

export default App;
```